```cpp
#pragma once
#include <Windows.h>

class AudioManager
{
    static AudioManager* s_pInstance;

    bool m_SoundOn;

    AudioManager()
        : m_SoundOn(true)
    {

    }

    //is this a singleton design pattern? one and only one instance?
    // global access, no ownership, lazyinitialisation
    // saves memory - but how?
    // Flexibility

public:
    static AudioManager* GetInstance()
    {
        if (s_pInstance == nullptr)
        {
            s_pInstance = new AudioManager();
        }
        return s_pInstance;
    }

    static void destroyInstance()
    {
        delete s_pInstance;
        s_pInstance = nullptr;
    }

    void ToggleSound()
    {
        m_SoundOn = !m_SoundOn;
    }

    bool IsSoundOn()
    {
        return m_SoundOn;
    }

    void playdoorclose()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(500, 75); // frequency and duration
        Beep(500, 75);
    }

    void playerdooropen()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(1397, 97);
    }

    void pickupkey()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(1568, 100);
    }

    void dropKeySound()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(1568, 200);
        Beep(1568, 50);
    }

    void moneySound()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(1568, 50);
    }

    void loseLife()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(200, 100);
    }

    void PlayLoseSound()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(500, 75);
        Beep(500, 75);
        Beep(500, 75);
        Beep(500, 75);
        Beep(500, 75);
        Beep(500, 75);
    }

    void win()
    {
        if (!m_SoundOn)
        {
            return;
        }
        Beep(1568, 200);
        Beep(1568, 200);
        Beep(1568, 200);
        Beep(1245, 1000);
        Beep(1397, 200);
        Beep(1397, 200);
        Beep(1397, 200);
        Beep(1175, 1000);
    }
};
```

```cpp
#include "AudioManager.h"

AudioManager* AudioManager::s_pInstance = nullptr;


```

```cpp
#include "PlaceableActor.h"

class Door : public PlaceableActor
{
public:
    Door(int x, int y, ActorColour colour, ActorColour closedColour);
    virtual void Draw() override;

    virtual ActorType GetType() override { return ActorType::Door; }
    bool IsOpen() { return m_isOpen; }
    void Open() { m_isOpen = true; }

private:
    bool m_isOpen;
    ActorColour m_closedColour;

};
```

```cpp
#include <iostream>
#include <Windows.h>
#include "Door.h"

Door::Door(int x, int y, ActorColour colour, ActorColour closedColour)
    :PlaceableActor(x, y, colour)
    , m_isOpen(false)
    , m_closedColour(closedColour)
{};

void Door::Draw()
{
    HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
    if (m_isOpen)
    {
        SetConsoleTextAttribute(console, (int)m_colour); // cast to an int
    }
    else
    {
        SetConsoleTextAttribute(console, (int)m_closedColour);
    }
    std::cout << "|";
    SetConsoleTextAttribute(console, (int)ActorColour::Regular);
}
```

```cpp
#include "PlaceableActor.h"

class Enemy : public PlaceableActor
{
public:
    Enemy(int x, int y, int deltaX = 0, int deltaY = 0);

    virtual ActorType GetType() override { return ActorType::Enemy; }
    virtual void Draw() override;
    virtual void Update() override;

private:

    int m_movementInX;
    int m_movementinY;

    int m_currentMovementX;
    int m_currentMovementY;

    int m_directionX;
    int m_directionY;

    void updateDirection(int& current, int& direction, int& movement);

};
```

```cpp
#include "Enemy.h"
#include <iostream>
#include <Windows.h>

Enemy::Enemy(int x, int y, int deltaX, int deltaY)
    : PlaceableActor(x, y, ActorColour::Green) // placing initial coordinates of enemy
    , m_currentMovementX(0)
    , m_currentMovementY(0)
    , m_directionX(0)
    , m_directionY(0)
    , m_movementInX(deltaX) //  The maximum distance the enemy can move in the x-direction
    , m_movementinY(deltaY) //  The maximum distance the enemy can move in the y-direction
{
    if (m_movementInX != 0)
    {
        m_directionX = 1;
    }
    if (m_movementinY != 0)
    {
        m_directionY = 1;
    }
}

void Enemy::Draw()
{
    HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(console, (int)m_colour);
    std::cout << (char)153; // prints coloured enemy.
    SetConsoleTextAttribute(console, (int)ActorColour::Regular);
}

void Enemy::Update() // update the state of the enemy
{
    if (m_movementInX != 0)
    {
        updateDirection(m_currentMovementX, m_directionX, m_movementInX);
    }
    if (m_movementinY != 0)
    {
        updateDirection(m_currentMovementY, m_directionY, m_movementinY);
    }

    this->SetXYPosition(m_pPosition->x + m_directionX, m_pPosition->y + m_directionY);
}

void Enemy::updateDirection(int& current, int& direction, int& movement) // responsible for handling the movement of the enemy
{
    current += direction;
    if (std::abs(current) > movement) // reverse movement. if we reach the
      end we want to loop back the other way.
    {
        current = movement * direction;
        direction *= -1; // change direction
    }
}

//  If the absolute value of the current movement becomes greater than the
  maximum allowed movement (movement), it means the enemy has reached the end
   of its allowed movement range
```

```cpp
#pragma once
#include "GameStateMachine.h"


class Game
{
public:
    GameStateMachine* m_pStateMachine;

public:
    Game();
    void Initialise(GameStateMachine* pStateMachine);
    void RunGameLoop();
    void Deinitialise();

private:
    bool Update(bool processInput = true);
    void Draw();

};

```

```cpp
#include "Game.h"

Game::Game()
    : m_pStateMachine(nullptr)
{};

void Game::Initialise(GameStateMachine* pStateMachine)
{
    if (pStateMachine)
    {
        pStateMachine->Init();
        m_pStateMachine = pStateMachine;
    }
};
void Game::RunGameLoop()
{
    bool isGameOver = false;
    while (!isGameOver)
    {
        Update(false);
        Draw();
        isGameOver = Update();
    }
    Draw();
};

void Game::Deinitialise()
{
    if (m_pStateMachine)
    {
        m_pStateMachine->CleanUp();
    }
};

bool Game::Update(bool processInput)
{
    return m_pStateMachine->UpdateCurrentState( processInput);
}

void Game::Draw()
{
    m_pStateMachine->DrawCurrentState();
}
```

```cpp
#pragma once
#include "GameState.h"
#include "Player.h"
#include "Level.h"
#include <Windows.h>
#include <vector>
#include <string>

class StateMachineExampleGame;

class GameplayState :
    public GameState
{
    StateMachineExampleGame* m_pOwner;

    Player m_player;
    Level* m_pLevel;

    bool m_beatLevel;
    int m_skipFrameCount;
    static constexpr int kFramesToSkip = 2;

    int m_currentLevel;
    vector<string> m_LevelNames;

public:
    GameplayState(StateMachineExampleGame* pOwner);
    ~GameplayState(); // clean up after levelnames
    virtual void Enter() override;
    virtual bool Update(bool processInput = true) override;
    virtual void Draw() override;

private:
    bool load();
    void HandleCollision(int newPlayerX, int newPlayerY);
    void DrawHUD(const HANDLE& console);

};


```

```cpp
#include "GameplayState.h"

#include <conio.h>
#include <iostream>
#include <assert.h>

#include "Enemy.h"
#include "Key.h"
#include "Door.h"
#include "Money.h"
#include "Goal.h"
#include "AudioManager.h"
#include "Game.h"
#include "Utility.h"

#include "StateMachineExampleGame.h"

using namespace std;

constexpr int kArrowInput = 224;
constexpr int kLeftArrow = 75;
constexpr int kRightArrow = 77;
constexpr int kUpArrow = 72;
constexpr int kDownArrow = 80;
constexpr int kEscapeKey = 27;
constexpr int kBackspace = 8;

GameplayState::GameplayState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
    , m_beatLevel(false)
    , m_skipFrameCount(0)
    , m_currentLevel(0)
    , m_pLevel(nullptr)
{
    m_LevelNames.push_back("Level4.txt");
    m_LevelNames.push_back("Level5.txt");
    m_LevelNames.push_back("Level6.txt");
}

GameplayState::~GameplayState()
{
    m_pLevel = nullptr;
    delete m_pLevel;
}

bool GameplayState::load()
{
    if (m_pLevel)
    {
        delete m_pLevel;
        m_pLevel = nullptr;
    }

    m_pLevel = new Level();

    return m_pLevel->LoadLevel(m_LevelNames.at(m_currentLevel), 
      m_player.GetXPositionPointer(), m_player.GetYPositionPointer());
}

void GameplayState::Enter()
{
    load();
}

bool GameplayState::Update(bool processInput)
{
    if (processInput && !m_beatLevel)
    {
        int input = _getch();
        int arrowInput = 0;
        int newPlayerX = m_player.GetXPosition();
        int newPlayerY = m_player.GetYPosition();

        // One of the Arrow keys were pressed
        if (input == kArrowInput)
        {
            arrowInput = _getch();
        }

        if ((input == kArrowInput && arrowInput == kRightArrow) ||
            ((char)input == 'd' || (char)input == 'D'))
        {
            newPlayerX++;
        }

        else if ((input == kArrowInput && arrowInput == kLeftArrow) ||
            ((char)input == 'a' || (char)input == 'A'))
        {
            newPlayerX--;
        }

        else if ((input == kArrowInput && arrowInput == kUpArrow) ||
            ((char)input == 'w' || (char)input == 'W'))
        {
            newPlayerY--;
        }

        else if ((input == kArrowInput && arrowInput == kDownArrow) ||
            ((char)input == 's' || (char)input == 'S'))
        {
            newPlayerY++;
        }

        else if (input == kEscapeKey)
        {
            m_pOwner->LoadScene
```

```cpp
              (StateMachineExampleGame::SceneName::MainMenu);
        }
        else if ((char)input == 'Z' || (char)input == 'z')
        {
            m_player.DropKey();
            AudioManager::GetInstance()->dropKeySound();
        }
        //If position never changed

        if (newPlayerX == m_player.GetXPosition() && newPlayerY == m_player.GetYPosition())
        {

        }
        else
        {
            HandleCollision(newPlayerX, newPlayerY);
        }
    }

    if (m_beatLevel)
    {
        ++m_skipFrameCount;
        if (m_skipFrameCount > kFramesToSkip) // player transitions over to X spot before sound.
        {
            m_beatLevel = false;
            m_skipFrameCount = 0;

            ++m_currentLevel;
            if (m_currentLevel == m_LevelNames.size())
            {
                Utility::WriteHighScore(m_player.GetMoney());
                AudioManager::GetInstance()->win();
                m_pOwner->LoadScene(StateMachineExampleGame::SceneName::Win);
            }
            else
            {
                load();
            }
        }
    }
    return false;
}

void GameplayState::HandleCollision(int newPlayerX, int newPlayerY) // more parameters to help with if loop
{
    bool isGameDone = false;
    PlaceableActor* collidedActor = m_pLevel->UpdateActors(newPlayerX, newPlayerY); // creates a placeable actor
    if (collidedActor != nullptr && collidedActor->IsActive())
    {
        switch (collidedActor->GetType())
        {
        case ActorType::Enemy:
        {
            Enemy* collidedEnemy = dynamic_cast<Enemy*>(collidedActor); // specifies the type/ thing we are trying to cast, in this case an enermy
            assert(collidedEnemy);
            AudioManager::GetInstance()->loseLife();
            // if the pointer is valid, if statement works, if it is a key none of the code will work
            collidedEnemy->Remove(); // if a collision with an enemy occurs, the enermy is removed.
            m_player.SetXYPosition(newPlayerX, newPlayerY); // players position is set to new position
            m_player.DecrementLives(); // decrmeent lives
            if (m_player.GetLive() < 0) // if less than zero game is over.
            {
                AudioManager::GetInstance()->PlayLoseSound();
                m_pOwner->LoadScene(StateMachineExampleGame::SceneName::Lose);
            }
            break;
        }
        case ActorType::Money:
        {
            Money* collidedMoney = dynamic_cast<Money*>(collidedActor); // if collided with money
            assert(collidedMoney);
            AudioManager::GetInstance()->moneySound();
            collidedMoney->Remove(); // remove the money
            m_player.AddMoney(collidedMoney->GetWorth()); // add the money and show the worth.
            m_player.SetXYPosition(newPlayerX, newPlayerY);
            break;
        }
        case ActorType::Key:
        {
            Key* collidedKey = dynamic_cast<Key*>(collidedActor); // returning null if fails within dynamic casts.
            assert(collidedKey);
            if (!m_player.HasKey())
            {
                m_player.PickUpKey(collidedKey);
                AudioManager::GetInstance()->pickupkey();
                collidedKey->Remove();
                m_player.SetXYPosition(newPlayerX, newPlayerY);
            }
            break;
        }
        case ActorType::Door:
        {
            Door* collidedDoor = dynamic_cast<Door*>(collidedActor);
            assert(collidedDoor);
            if (!collidedDoor->IsOpen())
            {
                if (m_player.HasKey(collidedDoor->GetColour()))
                {
                    collidedDoor->Open();
                    collidedDoor->Remove();
                    m_player.UseKey();
                    m_player.SetXYPosition(newPlayerX, newPlayerY);
                    AudioManager::GetInstance()->dropKeySound();
                }
                else
                {

                }
            }
            else
            {
                m_player.SetXYPosition(newPlayerX, newPlayerY); // player
                  goes through the door
            }
            break;
        }
        case ActorType::Goal:
        {
            Goal* collidedGoal = dynamic_cast<Goal*>(collidedActor);
            assert(collidedGoal);
            collidedGoal->Remove(); // removes actors
            m_player.SetXYPosition(newPlayerX, newPlayerY);
            m_beatLevel = true;
            break;
        }
        }
    }
    else if (m_pLevel->IsSpace(newPlayerX, newPlayerY)) // no collision
    {
        m_player.SetXYPosition(newPlayerX, newPlayerY);
    }
    else if (m_pLevel->IsWall(newPlayerX, newPlayerY))
    {
        // wall collision
    }
}

void GameplayState::Draw()
{
    HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
    system("cls");

    m_pLevel->Draw();

    //Set cursor position for player
```

```cpp
    COORD actorCursorPosition;
    actorCursorPosition.X = m_player.GetXPosition();
    actorCursorPosition.Y = m_player.GetYPosition();
    SetConsoleCursorPosition(console, actorCursorPosition);
    m_player.Draw();


    //Set cursor to end of level.
    COORD currentCursorPosition;
    actorCursorPosition.X = 0;
    actorCursorPosition.Y = m_pLevel->GetHeight();
    SetConsoleCursorPosition(console, actorCursorPosition);

    DrawHUD(console);
}

void GameplayState::DrawHUD(const HANDLE& console)
{
    cout << endl;

    // Top Border
    for (int i = 0; i < m_pLevel->GetWidth(); ++i)
    {
        cout << Level::WAL;
    }
    cout << endl;

    // left border

    cout << Level::WAL;

    cout << " wasd - move " << Level::WAL << " z - drop key " << Level::WAL;
    cout << "$: " << m_player.GetMoney() << " " << Level::WAL;
    cout << "Lives: " << m_player.GetLive() << " " << Level::WAL;
    cout << "Key: ";
    if (m_player.HasKey())
    {
        m_player.GetKey()->Draw();
    }
    else
    {
        cout << " ";
    }

    // right border

    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(console, &csbi);

    COORD pos;
    pos.X = m_pLevel->GetWidth() - 1;
    pos.Y = csbi.dwCursorPosition.Y;
    SetConsoleCursorPosition(console, pos);

    cout << Level::WAL;
    cout << endl;

    // Bottom Border
    for (int i = 0; i < m_pLevel->GetWidth(); ++i)
    {
        cout << Level::WAL;
    }
    cout << endl;
}
```

```cpp
#pragma once
// abstract class

class GameState
{

public:
    virtual ~GameState() = default;
    virtual void Enter() {};
    virtual bool Update(bool processInput = true) { return false; }
    virtual void Draw() = 0;
    virtual void Exit() {};



};
```

```cpp
#pragma once
// abstract that will create the GameStates from.

class GameState;

class GameStateMachine
{

public:

    virtual ~GameStateMachine() = default;

    virtual bool Init() = 0;
    virtual bool UpdateCurrentState(bool processInput = true) = 0;
    virtual void DrawCurrentState() = 0;
    virtual void ChangeState(GameState* pNewState) = 0;
    virtual void CleanUp() = 0;

};
```

```cpp
#include "PlaceableActor.h"

class Goal : public PlaceableActor
{
public:
    Goal(int x, int y);

    virtual ActorType GetType() override { return ActorType::Goal; }
    virtual void Draw() override;

};
```

```
#include <iostream>
#include "Goal.h"

Goal::Goal(int x, int y)
    : PlaceableActor(x, y)
{

}

void Goal::Draw()
{
    std::cout << "X";
}
```

```cpp
#pragma once

#include "GameState.h"
#include <set>

class StateMachineExampleGame;


class HighScoreState :
    public GameState
{
    StateMachineExampleGame* m_pOwner;
    std::set<int> m_highscore;

public:
    HighScoreState(StateMachineExampleGame* pOwner);
    ~HighScoreState() = default;

    virtual bool Update(bool processInput = true) override;
    virtual void Draw() override;


};


```

```cpp
#include "HighScoreState.h"

#include <iostream>
#include <conio.h>

#include "StateMachineExampleGame.h"
#include "Utility.h"

HighScoreState::HighScoreState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
{
    m_highscore = Utility::WriteHighScore(0);
}

bool HighScoreState::Update(bool processInput)
{
    if (processInput)
    {
        int input = _getch();
        m_pOwner->LoadScene(StateMachineExampleGame::SceneName::MainMenu);
    }
    return false;
}

void HighScoreState::Draw()
{
    system("cls");
    cout << endl << endl << endl;
    cout << "             - - - - HIGH SCORES - - - -            " << endl << endl;

    for (auto i = m_highscore.rbegin(); i != m_highscore.rend(); ++i)
    {
        cout << "            " << *i << endl;
    }

    cout << endl;
    cout << endl;
    cout << "Press any key to go back to the main menu. " << endl << endl;
}
```

```
#include "PlaceableActor.h"

class Key : public PlaceableActor
{
public:
    Key(int x, int y, ActorColour colour)
        : PlaceableActor(x, y, colour)
    {
    }

    virtual ActorType GetType() override { return ActorType::Key; }
    virtual void Draw() override;
};
```

```
#include <iostream>
#include <Windows.h>
#include "Key.h"

void Key::Draw()
{
    HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(console, (int)m_colour);
    std::cout << "+"; // prints coloured key.
    SetConsoleTextAttribute(console, (int)ActorColour::Regular);
}
```

```
#include "Player.h"
#include <string>
#include <vector>
using namespace std;

class PlaceableActor;


class Level
{
    char* plevel;
    int height;
    int width;

    vector<PlaceableActor*> m_pActors;

public:
    Level();
    ~Level();

    bool LoadLevel(string levelName, int* playerX, int* playerY);
    void Draw();
    PlaceableActor* UpdateActors(int x, int y);

    bool IsSpace(int x, int y);
    bool IsWall(int x, int y);

    int GetHeight() { return height; }
    int GetWidth() { return width; }
    int GetIndex(int x, int y);

    static constexpr char WAL = (char)219;

private:
    bool Convert(int* playerX, int* playerY);

};

```

```cpp
#include <Windows.h>
#include "Level.h"
#include <iostream>
#include <fstream>
#include "Player.h"
#include "Enemy.h" // derived Placeable Actor Classes
#include "Key.h"
#include "Door.h"
#include "Goal.h"
#include "Money.h"
#include <assert.h>

using namespace std;

Level::Level()
    : plevel(nullptr)
    , height(0)
    , width(0)
{

};

Level::~Level()
{
    if (plevel != nullptr)
    {
        delete[] plevel;
        plevel = nullptr;
    }

    while (!m_pActors.empty())
    {
        delete m_pActors.back(); // return us the elements at end, then delete
        m_pActors.pop_back(); // continue to delete the remaining vector elements.
    }
};

bool Level::LoadLevel(string levelName, int* playerX, int* playerY)
{
    levelName.insert(0, "../");
    ifstream levelFile;
    levelFile.open(levelName);
    if (!levelFile)
    {
        cout << "An error has occured." << endl;
        return false;
    }
    else
    {
        constexpr int tempSize = 25;
        char temp[tempSize];
```

```
52
53          levelFile.getline(temp, tempSize, '\n');
54          width = atoi(temp); // converts integer into width.
55
56          levelFile.getline(temp, tempSize, '\n'); // line 83 and line 87 link.
57          height = atoi(temp);
58
59          plevel = new char[width * height]; // array that we need to deallocate.
60          levelFile.read(plevel, width * height);
61
62          if (playerX != nullptr && playerY != nullptr)
63          {
64              bool anyWarnings = Convert(playerX, playerY);
65              if (anyWarnings)
66              {
67                  cout << "There are some warnings in the level data. see above." << endl;
68                  system("pause");
69              }
70          }
71          return true;
72      }
73  }
74
75  void Level::Draw()
76  {
77      HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE); // temprary variables being deleted at the end of draw.
78      SetConsoleTextAttribute(console, (int)ActorColour::Regular);
79
80      //Draw Level
81      for (int y = 0; y < GetHeight(); ++y)
82      {
83          for (int x = 0; x < GetWidth(); ++x)
84          {
85              int indexToPoint = GetIndex(x, y);
86              cout << plevel[indexToPoint];
87          }
88          cout << endl;
89      }
90
91      COORD actorCursorPosition; // position the cursor at correct location, x and y variables
92
93      // Draw actors
94
95      for (auto actor = m_pActors.begin(); actor != m_pActors.end(); ++actor) // going to the beginning and through the end.
96      {
97          if ((*actor)->IsActive()) // if active we want to draw.
98          {
```

```
99              actorCursorPosition.X = (*actor)->GetXPosition();
100             actorCursorPosition.Y = (*actor)->GetYPosition();
101             SetConsoleCursorPosition(console, actorCursorPosition); // set position manually to this point.
102             (*actor)->Draw(); // draw the actors, tempoary variable in line 93 is now finished and deleted from the stack.
103         }
104     }
105 }
106 
107 bool Level::IsSpace(int x, int y)
108 {
109     return plevel[GetIndex(x, y)] == ' ';
110 }
111 bool Level::IsWall(int x, int y)
112 {
113     return plevel[GetIndex(x, y)] == WAL;
114 }
115 
116 bool Level::Convert(int* playerX, int* playerY)
117 {
118     bool anyWarnings = false;
119 
120     for (int y = 0; y < height; ++y)
121     {
122         for (int x = 0; x < width; ++x)
123         {
124             int intIndex = GetIndex(x, y);
125 
126             switch (plevel[intIndex])
127             {
128             case '+':
129             case '-':
130             case '|':
131             {
132                 plevel[intIndex] = WAL;
133                 break;
134             }
135             case ' ':
136             {
137                 break;
138             };
139             case 'r':
140                 plevel[intIndex] = ' ';
141                 m_pActors.push_back(new Key(x, y, ActorColour::Red));
142                 break;
143             case 'g':
144                 plevel[intIndex] = ' ';
145                 m_pActors.push_back(new Key(x, y, ActorColour::Green));
146                 break;
147             case 'b':
148                 plevel[intIndex] = ' ';
149                 m_pActors.push_back(new Key(x, y, ActorColour::Blue));
```

```
                    break;
                case 'R':
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Door(x, y, ActorColour::Red, 
                      ActorColour::RedSolid));
                    break;
                case 'G':
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Door(x, y, ActorColour::Green, 
                      ActorColour::GreenSolid));
                    break;
                case 'B':
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Door(x, y, ActorColour::Blue, 
                      ActorColour::BlueSolid));
                    break;
                case 'X':
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Goal(x, y));
                    break;
                case '$':
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Money(x, y, 1 + rand() % 5));
                    break;
                case '@':
                {
                    plevel[intIndex] = ' ';
                    if (playerX != nullptr && playerY != nullptr)
                    {
                        *playerX = x;
                        *playerY = y;
                    }
                    break;
                }
                case 'e':
                    m_pActors.push_back(new Enemy(x, y));
                    plevel[intIndex] = ' '; // clear level
                    break;
                case 'h': // horiztonal enemy
                    m_pActors.push_back(new Enemy(x, y, 3, 0));
                    plevel[intIndex] = ' ';
                    break;
                case 'v': // vertical enemy
                    plevel[intIndex] = ' ';
                    m_pActors.push_back(new Enemy(x, y, 0, 2));
                    plevel[intIndex] = ' ';
                    break;
                default:
                {
                    cout << "Invalid character in file " << plevel[intIndex] << 
                      endl;
                    anyWarnings = true;
                    break;
                }
                }
            }
        }
        return anyWarnings;
    }

    int Level::GetIndex(int x, int y)
    {
        return x + y * width;
    }

    // Updates all actors and returns a colliding actor is there is one.

    PlaceableActor* Level::UpdateActors(int x, int y ) // pass in x and y of 
      player.
    {

        PlaceableActor* collidedActor = nullptr;

        for (auto actor = m_pActors.begin(); actor != m_pActors.end(); ++actor)
        {
            (*actor)->Update(); //update all actors

            if (x == (*actor)->GetXPosition() && y == (*actor)->GetYPosition
              ()) // collision occured
            {
                assert(collidedActor == nullptr); // if assertion fails, two 
                  points have met.
                collidedActor = (*actor); // points to the location of the 
                  collision.
            }
        }
        return collidedActor;
    }
```

```
#pragma once

#include "StateMachineExampleGame.h"

#include "GameState.h"
class LoseState :
    public GameState
{
        StateMachineExampleGame * m_pOwner;

    public:
        LoseState(StateMachineExampleGame* pOwner);
        ~LoseState() = default;

        virtual bool Update(bool processInput = true) override;
        virtual void Draw() override;
};


```

```cpp
#include "LoseState.h"

#include <iostream>
#include <conio.h>

#include "StateMachineExampleGame.h"

using namespace std;

LoseState::LoseState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
{}

bool LoseState::Update(bool processInput)
{
    if (processInput)
    {
        int input = _getch();
        m_pOwner->LoadScene(StateMachineExampleGame::SceneName::MainMenu);
    }
    return false;
}

void LoseState::Draw()
{
    system("cls");
    cout << endl << endl << endl;
    cout << "             - - - - GAME OVER - - - -            " << endl << endl;
    cout << "             BETTER LUCK NEXT TIME        " << endl << endl;
    cout << "             PRESS ANY KEY TO GO BACK TO MAIN MENU        " << endl << endl;

}
```

```
#pragma once
#include "GameState.h"

class StateMachineExampleGame;

class MainMenuState :
    public GameState
{
    StateMachineExampleGame* m_pOwner;

public:
    MainMenuState(StateMachineExampleGame* pOwner);
    ~MainMenuState() = default;

    virtual bool Update(bool processInput = true) override;
    virtual void Draw() override;

};


```

```cpp
#include "MainMenuState.h"

#include <iostream>
#include <conio.h>

#include "StateMachineExampleGame.h"

using namespace std;

constexpr int kEscape = 27;

constexpr char kPlay = '1';
constexpr char kHighScore = '2';
constexpr char kSettings = '3';

constexpr char kQuit = '4';

MainMenuState::MainMenuState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
{}

bool MainMenuState::Update(bool processInput)
{
    bool shouldQuit = false;
    if (processInput)
    {
        int input = _getch();
        if (input == kEscape || char(input) == kQuit)
        {
            shouldQuit = true;
        }
        else if ((char)input == kPlay)
        {
            m_pOwner->LoadScene
              (StateMachineExampleGame::SceneName::Gameplay);
        }
        else if ((char)input == kHighScore)
        {
            m_pOwner->LoadScene
              (StateMachineExampleGame::SceneName::Highscore);
        }
        else if ((char)input == kSettings)
        {
            m_pOwner->LoadScene
              (StateMachineExampleGame::SceneName::Settings);
        }
    }
    return shouldQuit;
}

void MainMenuState::Draw()
{
    system("cls");
    cout << endl << endl << endl;
    cout << "            - - - - MAIN MENU - - - -           " << endl << endl;
    cout << "                 " << kPlay << ". Play " << endl;
    cout << "                 " << kHighScore << ". Highscore " << endl;
    cout << "                 " << kSettings << ". Settings " << endl;
    cout << "                 " << kQuit << ". Quit " << endl;

}
```

```cpp
#include "PlaceableActor.h"

class Money : public PlaceableActor
{
public:
    Money(int x, int y, int worth);

    int GetWorth() const { return m_worth; }

    virtual ActorType GetType() override { return ActorType::Money; }
    virtual void Draw() override;

private:
    int m_worth;

};
```

```cpp
#include "Money.h"
#include <iostream>

Money::Money(int x, int y, int worth)
    : PlaceableActor(x, y)
    , m_worth(worth)
{

}

void Money::Draw()
{
    std::cout << "$";
}
```

```
#ifndef PLACEABLEACTOR_H
#define PLACEABLEACTOR_H
#include "Point.h"

enum class ActorColour
{
    Regular = 7,
    Blue = 9,
    Green = 10,
    Red = 12,
    GreenSolid = 34,
    RedSolid = 255,
    BlueSolid = 153
};

enum class ActorType
{
    Door,
    Enemy,
    Goal,
    Key,
    Money,
    Player
};

class PlaceableActor
{
public:
    PlaceableActor(int x, int y, ActorColour colour = ActorColour::Regular);
    virtual ~PlaceableActor();

    int GetXPosition();
    int GetYPosition();
    int* GetXPositionPointer();
    int* GetYPositionPointer();
    void SetXYPosition(int x, int y);

    ActorColour GetColour() { return m_colour; }

    void Remove() { m_IsActive = false; }
    bool IsActive() { return m_IsActive; }
    void Place(int x, int y);

    virtual ActorType GetType() = 0;
    virtual void Draw() = 0;
    virtual void Update() // some placeable actors will not need to update themselves
    {

    }

protected:
    Point* m_pPosition;
    bool m_IsActive;
    ActorColour m_colour;

};

#endif
```

```cpp
#include "PlaceableActor.h"

PlaceableActor::PlaceableActor(int x, int y, ActorColour colour)
    : m_pPosition(new Point(x, y))
    , m_IsActive(true),
    m_colour(colour)
{

}

PlaceableActor::~PlaceableActor()
{
    delete m_pPosition;
    m_pPosition = nullptr;
}

int PlaceableActor::GetXPosition()
{
    return m_pPosition->x;
}

int PlaceableActor::GetYPosition()
{
    return m_pPosition->y;
}

int* PlaceableActor::GetXPositionPointer()
{
    return &(m_pPosition->x);
}

int* PlaceableActor::GetYPositionPointer()
{
    return &(m_pPosition->y);
}

void PlaceableActor::SetXYPosition(int x, int y)
{
    m_pPosition->x = x;
    m_pPosition->y = y;
}

void PlaceableActor::Place(int x, int y)
{
    m_pPosition->x = x;
    m_pPosition->y = y;
    m_IsActive = true;
}
```

```
#ifndef _PLAYER_H_
#define _PLAYER_H_

#include "PlaceableActor.h"

class Key; // you can only forward declare pointer types, specific items

class Player : public PlaceableActor
{

public:
    Player();

    bool HasKey();
    bool HasKey(ActorColour colour);
    void PickUpKey(Key* key);
    void UseKey();
    void DropKey();
    Key* GetKey() { return m_pCurrentKey; }

    // nothing in the class is using key in a functions

    void AddMoney(int money) { m_money += money; }
    int GetMoney() { return m_money; }

    int GetLive() { return m_lives; }
    void DecrementLives() { m_lives--; }

    virtual ActorType GetType() override { return ActorType::Player; }
    virtual void Draw() override;

private:
    Key* m_pCurrentKey;
    int m_money;
    int m_lives;

};

#endif // !_PLAYER_H_

```

```cpp
#include "Player.h"
#include "Key.h" // using key in a function
#include <iostream>

using namespace std;

constexpr int kStartNumberOfLives = 1;

Player::Player()
    : PlaceableActor(0,0)
    , m_pCurrentKey(nullptr)
    , m_money(0)
    , m_lives(kStartNumberOfLives)
{
};

bool Player::HasKey()
{
    return m_pCurrentKey != nullptr;
}

bool Player::HasKey(ActorColour colour)
{
    return HasKey() && m_pCurrentKey->GetColour() == colour;
}

void Player::PickUpKey(Key* key)
{
    m_pCurrentKey = key;
}

void Player::UseKey()
{
    m_pCurrentKey->Remove();
    m_pCurrentKey = nullptr;
}

void Player::DropKey()
{
    if (m_pCurrentKey)
    {
        m_pCurrentKey->Place(m_pPosition->x, m_pPosition->y);
        m_pCurrentKey = nullptr;
    }
}

void Player::Draw()
{
    cout << "@";
}
```

```cpp

struct Point
{
    int x;
    int y;

    Point()
        : x(0)
        , y(0)
    {

    }

    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

};




```

```cpp
#include <iostream>
#include <conio.h>
#include <Windows.h>
#include <fStream>

#include "StateMachineExampleGame.h"
#include "AudioManager.h"
#include "Game.h"
using namespace std;

int main() {

    Game myGame;

    StateMachineExampleGame gameStateMachine(&myGame);

    myGame.Initialise(&gameStateMachine);
    myGame.RunGameLoop();
    myGame.Deinitialise();

    AudioManager::destroyInstance();

    return 0;

}


```

```cpp
#pragma once
#include "GameState.h"

class StateMachineExampleGame;

class SettingState :
    public GameState
{
    StateMachineExampleGame* m_pOwner;

public:
    SettingState(StateMachineExampleGame* pOwner);
    ~SettingState() = default;

    virtual bool Update(bool processInput = true) override;
    virtual void Draw() override;

};


```

```cpp
#include "SettingState.h"

#include <iostream>
#include <conio.h>

#include "StateMachineExampleGame.h"
#include "AudioManager.h"

using namespace std;

constexpr int kEscape = 27;

constexpr char kSound = '1';
constexpr char kMainMenu = '2';

SettingState::SettingState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
{}

bool SettingState::Update(bool processInput)
{
    if (processInput)
    {
        int input = _getch();
        if (input == kEscape || char(input) == kMainMenu)
        {
            m_pOwner->LoadScene(StateMachineExampleGame::SceneName::MainMenu);
        }
        else if ((char)input == kSound)
        {
            AudioManager::GetInstance()->ToggleSound();
            AudioManager::GetInstance()->moneySound();
        }
    }
    return false;
}

void SettingState::Draw()
{
    system("cls");
    cout << endl << endl << endl;
    cout << "             - - - - SETTINGS - - - -            " << endl;
    cout << "                  " << kSound << ". Play " << endl;
    cout << "                  " << "Toggle Sound: ";
    if (AudioManager::GetInstance()->IsSoundOn())
    {
        cout << "ON" << endl;
    }
    else
    {
        cout << "OFF" << endl;
    }
    cout << "                    " << kMainMenu << ". Back to Main Menu " <<  ↩
      endl;

}
```

```cpp
#include "StateMachineExampleGame.h"

#include "MainMenuState.h"
#include "GameplayState.h"
#include "SettingState.h"
#include "HighScoreState.h"
#include "WinState.h"
#include "LoseState.h"

#include "Game.h"

StateMachineExampleGame::StateMachineExampleGame(Game* pOwner)
    : m_pOwner(pOwner)
    , m_pCurrentState(nullptr)
    , m_pNewState(nullptr)
{};

bool StateMachineExampleGame::Init()
{
    LoadScene(SceneName::MainMenu);
    return true;
}

bool StateMachineExampleGame::UpdateCurrentState(bool processInput)
{
    bool done = false;
    if (m_pNewState != nullptr)
    {
        ChangeState(m_pNewState);
        m_pNewState = nullptr;
    }

    if (m_pCurrentState != nullptr)
    {
        done = m_pCurrentState->Update(processInput);
    }
    return done;
}

void StateMachineExampleGame::DrawCurrentState()
{
    if (m_pCurrentState)
    {
        m_pCurrentState->Draw();
    }
}

void StateMachineExampleGame::ChangeState(GameState* pNewState)
{
    if (m_pCurrentState)
    {
        m_pCurrentState->Exit();
    }

    delete m_pCurrentState;
    m_pCurrentState = pNewState;
    pNewState->Enter();
}

void StateMachineExampleGame::CleanUp()
{
    if (m_pCurrentState)
    {
        m_pCurrentState->Exit();
        delete m_pCurrentState;
    }
}

void StateMachineExampleGame::LoadScene(SceneName scene)
{
    switch (scene)
    {
    case SceneName::MainMenu:
        m_pNewState = new MainMenuState(this);
        break;
    case SceneName::Gameplay:
        m_pNewState = new GameplayState(this);
        break;
    case SceneName::Settings:
        m_pNewState = new SettingState(this);
        break;
    case SceneName::Highscore:
        m_pNewState = new HighScoreState(this);
        break;
    case SceneName::Win:
        m_pNewState = new WinState(this);
        break;
    case SceneName::Lose:
        m_pNewState = new LoseState(this);
        break;
    default:
        break;
    }
}
```

```
#pragma once
#include "GameStateMachine.h"

class Game;
class GameState;

class StateMachineExampleGame :
    public GameStateMachine
{
public:
    enum class SceneName
    {
        None,
        MainMenu,
        Gameplay,
        Settings,
        Highscore,
        Lose,
        Win
    };

private:
    Game* m_pOwner;

    GameState* m_pCurrentState;
    GameState* m_pNewState;

public:
    StateMachineExampleGame(Game* pOwner);

    virtual bool Init() override;
    virtual bool UpdateCurrentState(bool processInput = true) override;
    virtual void DrawCurrentState() override;
    virtual void ChangeState(GameState* pNewState) override;
    virtual void CleanUp() override;
    void LoadScene(SceneName scene);

};

```

```cpp
#pragma once

#include <iostream>
#include <set>
#include <string>
#include <fstream>
#include <iterator>

using namespace std;

class Utility
{
public:
    static set<int> WriteHighScore(int score)
    {
        // see if file exists and read values

        string fileName = "highscores.txt";
        ifstream highScoreFile(fileName);
        istream_iterator<int> start(highScoreFile), end;
        set<int> highscores(start, end);
        highScoreFile.close();

        // if its empty, populate and save it.

        if (highscores.size() == 0)
        {
            highscores.insert(100);
            highscores.insert(50);
            highscores.insert(20);
            highscores.insert(10);
            highscores.insert(5);

            ofstream outFile(fileName);
            ostream_iterator<int> output_iterator(outFile, "\n");
            copy(highscores.begin(), highscores.end(), output_iterator);
            outFile.close();
        }

        // write score

        highscores.insert(score);

        // remove lowest score

        highscores.erase(highscores.begin());

        // write the highscores.
        ofstream outFile(fileName);
        ostream_iterator<int> output_iterator(outFile, "\n");
        copy(highscores.begin(), highscores.end(), output_iterator);
        outFile.close();

        return highscores;
    }


};
```

```cpp
#pragma once
#include "GameState.h"

#include "StateMachineExampleGame.h"

class WinState :
    public GameState
{
    StateMachineExampleGame* m_pOwner;

public:
    WinState(StateMachineExampleGame* pOwner);
    ~WinState() = default;

    virtual bool Update(bool processInput = true) override;
    virtual void Draw() override;

};


```

```cpp
#include "WinState.h"

#include <iostream>
#include <conio.h>

#include "StateMachineExampleGame.h"

using namespace std;

WinState::WinState(StateMachineExampleGame* pOwner)
    : m_pOwner(pOwner)
{}

bool WinState::Update(bool processInput)
{
    if (processInput)
    {
        int input = _getch();
        m_pOwner->LoadScene(StateMachineExampleGame::SceneName::MainMenu);
    }
    return false;
}

void WinState::Draw()
{
    system("cls");
    cout << endl << endl << endl;
    cout << "             - - - - WELL DONE - - - -            " << endl << endl;
    cout << "             YOU BEAT THE GAME.       " << endl << endl;
    cout << "                                          " << endl << endl;

}
```